Naval Research Laboratory

Stennis Space Center, MS 39529-5004

Parallel Implementation of the QUODDY 3-D Finite-Element Circulation Model

Timothy J. Campbell

Mississippi State University
NAVOCEANO MSRC PET

Cheryl Ann Blain

Ocean Dynamics and Prediction Branch
Oceanography Division

June 28, 2002

REPORT DOCUMENTATION PAGE

1. REPORT DATE (DD-MM-YYYY) June 28, 2002	2. REPORT TYPE	3. DATES COVERED (From - To)

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Parallel Implementation of the Quoddy 3-D Finite Element Circulation Model	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER PE - 0602435N

6. AUTHOR(S)	5d. PROJECT NUMBER
Timothy J. Campbell and Cheryl Ann Blain	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Naval Research Laboratory Oceanography Division Stennis Space Center, MS 39529-5004	NRL/FR/7320--02-10,021

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR / MONITOR'S ACRONYM(S)
Office of Naval Research 800 North Quincy Street Arlingtonn, VA 22217-5660	11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

Naval Research Laboratory Report

14. ABSTRACT

This report describes implementation of the QUODDY finite-element circulation model on shared-memory multiprocessor computers using OpenMP. Because all code modifications were restricted to the main computational routines and no changes are required in the user interface and configuration files, the parallel code can be seamlessly integrated into existing regional applications of the model. Bit-for-bit matching between serial and parallel execution has been achieved. The code modifications reduced the execution time per model time step of one test case from 21.1 s on a single processor to about 1.4 s on 32 processors. By reducing turnaround time and enabling substantial increases in model resolution, the parallel code will benefit further coastal ocean circulation model development.

15. SUBJECT TERMS

Parallel computing, Finite-element, Coastal ocean circulation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Cheryl Ann Blain
a. REPORT	b. ABSTRACT	c. THIS PAGE	UL	34	19b. TELEPHONE NUMBER (include area code) (228) 688-5450
Unclassified	Unclassified	Unclassified			

CONTENTS

PARALLEL IMPLEMENTATION OF THE QUODDY 3-D FINITE-ELEMENT CIRCULATION MODEL

1. INTRODUCTION

Realistic representations of coastal ocean circulation require the use of three-dimensional (3-D) numerical models. These models contain multiple equations that have a significant number of unknown variables, e.g., water level, three velocity components, temperature, salinity, and turbulence-related quantities, whose solutions translate into potentially large costs when using only a single processor. Applications in coastal areas where grid refinement is high and/or grid boundaries are located in offshore waters result in computational domains that can be rather large – further exacerbating the computational overhead of 3-D simulations. The need for a multiprocessor computational capability is clear when dealing with 3-D coastal circulation models.

One such coastal circulation model is QUODDY, the 3-D ocean circulation model that is part of the Dartmouth College suite of models [1]. The QUODDY model has had tremendous success in studies focused largely on continental shelf circulation [2], but such applications have been purposefully limited in resolution and domain size due to computational constraints. Modifications to the QUODDY software that permit the use of multiprocessor computational resources will dramatically change the way in which this model is applied to coastal circulation problems. Coastal circulation model development should also proceed more rapidly since the turn-around time for a model application is reduced.

We chose to port the QUODDY model to shared-memory multiprocessor computers using the OpenMP multithreading directives. These can provide a minimally intrusive and incremental method for producing a parallel code. Appendix A briefly describes the OpenMP programming model. With this approach, we have been able to produce a moderately scalable code that requires no change to the user interface and configuration files. This report describes our development of a scalable version of QUODDY using OpenMP. The necessary code modifications are documented to provide a record for those who maintain the QUODDY model. Section 2 briefly describes the QUODDY model; Section 3 provides an execution profile of the original serial code. Section 4 provides details on the development of the parallel version of QUODDY. Section 5 provides details on how the performance of the parallel code and how the code changes were verified. Performance comparisons with alternate approaches using OpenMP are described in Section 6, and a summary is given in Section 7.

2. DESCRIPTION OF THE QUODDY MODEL

The QUODDY model, developed by the Numerical Methods Laboratory at Dartmouth College, represents the most physically advanced finite-element circulation model to date [1]. The QUODDY

model is a time marching simulator based on the 3-D hydrodynamic equations subject to conventional Boussinesq and hydrostatic assumptions. A wave-continuity form of the mass conservation equation [3,4], designed to eliminate numerical noise at or below two times the grid spacing, is solved in conjunction with momentum conservation and transport equations for temperature and salinity. Vertical mixing is represented with a level 2.5 turbulence closure [5]. This turbulence closure scheme accounts for processes occurring over the vertical extent of the water column such as diffusion, shear production, buoyancy, production, and dissipation. Variable horizontal resolution is provided on unstructured triangular meshes. A general terrain-following vertical coordinate allows smooth resolution of surface and bottom boundary layers. The QUODDY model is dynamically equivalent to the often used Princeton Ocean Model [6]. The advantage of the current model lies in its finite-element formulation that allows for greater flexibility in representing geometric complexity and strong horizontal gradients in either bathymetry and/or velocity.

The work described in this report pertains to Version 5 Release 1.0 of QUODDY (hereafter referred to as QUODDY5). The QUODDY5 software is written in ANSI Fortran 77 and consists of the following six program files and header file.

quoddy5_1.0_main.f: Main program for QUODDY5;

quoddy5_1.0_coresubs.f: Core subroutines for QUODDY5;

quoddy5_1.0_usrsubs_resources.f: Supporting subroutines for QUODDY5 user-specified subroutines;

quoddy5_1.0_usrsubs.f: User-specified subroutines for QUODDY5;

DCMSPAK_000607.f: Routines from Dartmouth Circulation Models Software (Equation of State routines, Baroclinic Pressure Gradient routines, routines from FUNDY6);

NMLPAKS_000607.f: Selected packages from the Dartmouth Numerical Methods Library (FEMPAK, GOMPAK, IOSPAK, MAXPAK, and SPRSPAK);

DCMS.DIM: Header file with parameters for various Dartmouth Circulation Models Software.

The user-specified subroutines in quoddy5_1.0_usrsubs.f are built with a standardized interface. These routines are used to specify physical forcing, vertical meshing, boundary conditions, and the manner in which results are to be analyzed and written. The exact makeup of the user-specified routines depends on the user and the region of application.

The main computational (time-stepping) loop of QUODDY5 can be described with four logical sections, as illustrated in Fig. 1. Within each logical section, the numerics are carried out through a combination of subroutine calls and operations local to the time-stepping loop. Figure 1 lists the subroutines called within each section. For reference, Appendix B gives the Fortran source code for the time-stepping loop (with OpenMP modifications).

3. PROFILE OF SERIAL CODE

Prior to making modifications for OpenMP it is useful to generate an execution profile and to identify code regions where the most time is spent and which routines called which other routines during execution. The execution profile for QUODDY5 is given in Table 1; only the subroutines from the time-stepping loop are listed.[1] The profile was performed over 10 time steps, with temperature and salinity transport enabled, on a finite-element mesh with 17440 horizontal and 51

---

[1]In this work, a commonly available profiling application known as gprof was used. Gprof counts the number of times a routine is called and estimates the amount of time spent in each routine using a sampling process. Because of the

1. Setup for present time level K
   - Load atmospheric forcing: ATMOSQ5
   - Load point source information: POINTSOURCEQ5
   - Compute linearized bottom stress coefficients: QUADSTRESS
   - Evaluate baroclinic pressure gradients: RHOXYQ4
   - Evaluate horizontal eddy viscosity/diffusivity: SMAGOR1
   - Evaluate nonlinear advection and horizontal diffusion of momentum: SPRSMLTIN2, SPRSCONV, CONVECTION
2. Solve wave equation for free surface elevation: ELEVATIONQ5
3. Solve for vertical structure of the 3-D dependent variables: VERTICALQ5
4. Update arrays and increment timing parameters
   - Store present information as time level K-1
   - Compute equation of state: EQSTATE_2D
   - Compute vertical velocities: SPRSCONV, VERTVEL3_2, VERTAVG

Fig. 1 — Time stepping loop of QUODDY5 with logical sections enumerated. The subroutines called within each section are also listed.

vertical nodes. The numbers in the first three columns express the time spent as a percentage of the total execution time. The total execution time includes both the pre-time-stepping section and the time stepping loop. However, the pre-time-stepping section is a small fraction of the total time (less than 2%). Columns 2 and 3 separate the total time in column 1 into the time used only by the subroutine itself (column 2) and the time used by that subroutine's descendents (column 3).

Table 1 shows that almost 50% of the execution time is spent in the subroutine VERTICALQ5 (which is called once every time step) and its descendents. The execution time used by VERTICALQ5 is dominated by a single loop over the horizontal nodes (of the triangular mesh). In this loop, the vertical structure is computed for vertical grid points directly under each horizontal node. More detail on the profile for subroutine VERTICALQ5 is given in Table 2. All the descendents of VERTICALQ5 with the exception of SPRSMLT are called from within the vertical structure loop. Because the vertical data under a horizontal node does not depend on information from neighboring horizontal nodes, each loop iteration is independent. Thus, the vertical structure loop can easily be done in parallel.

Among the subroutines called once per time step that use the most time, ELEVATIONQ5 is second after VERTICALQ5. From the profile of ELEVATIONQ5 given in Table 2, we see that most of the time is spent in descendents BANSOLTR and VERTGRIDQ5. As it turns out, the routines called by VERTGRIDQ5 (that set up the vertical grid spacing) consist of loops over horizontal nodes that can be done in parallel. However, the matrix solve in BANSOLTR for the sea surface elevation cannot be done in parallel without extensive modifications beyond loop parallel constructs. At this point, it is not clear if modifications to the matrix solve would give any benefit.

sampling process, the timings are subject to statistical inaccuracy. Additional inaccuracy of timings is caused by the overhead of profiling routines invoked during execution.

Table 1 — Profile for MAIN of QUODDY5

Percent of total time			Number of	Parent
Total	Self	Descendents	times called	Children
100.0	8.2	91.8	1	MAIN
49.7	29.5	20.2	10	VERTICALQ5
8.7	8.7	0.0	348800	SPRSCONV
5.9	0.9	5.0	10	ELEVATIONQ5
5.8	5.8	0.0	174400	CONVECTION
5.6	5.4	0.2	11	RHOXYQ4
4.5	3.6	0.9	174400	VERTVEL3_2
3.0	2.2	0.8	10	SMAGOR1
3.0	3.0	0.0	174400	SPRSMLTIN2
2.9	2.9	0.0	20	VERTAVG
1.4	1.4	0.0	11	EQSTATE2_2D
0.3	0.3	0.0	11	VERTSUM
0.0	0.0	0.0	11	POINTSOURCEQ5
0.0	0.0	0.0	11	ATMOSQ5
0.0	0.0	0.0	12	SPRSINVMLT
0.0	0.0	0.0	10	QUADSTRESS

The 0.9% time used by ELEVATIONQ5 itself involves setting up the right-hand side of the matrix equation that is solved for the sea surface elevation. Most of that time is spent in a single loop over the elements of the horizontal triangular mesh. At each iteration of the element loop, data at each of the three nodes associated with that particular element are modified. Because nodes are shared by multiple elements, this results in data dependencies that prevent executing the element loop in parallel. In other words, two threads processing different elements (iterations) that share a node will try to modify the data at that node at the same time with no guarantee of correctness. This same issue also occurs in the subroutine SMAGOR1 which, according to gprof, consumes 3.0% of the execution time.

Table 2 — Profile for subroutines VERTICALQ5 and ELEVATIONQ5

Percent of total time			Number of	Parent
Total	Self	Descendents	times called	Children
49.7	29.5	20.2	10	VERTICALQ5
9.8	9.8	0.0	697600	SPRSMLTIN3
3.6	3.6	0.0	697600	THOMAS
3.4	3.4	0.0	174400	GALPERINQ5
2.4	2.4	0.0	174400	ELEMCOEFS
0.9	0.9	0.0	174610	CTHOMAS
0.1	0.1	0.0	20	SPRSMLT
0.0	0.0	0.0	630	VERTINTI
5.9	0.9	5.0	10	ELEVATIONQ5
3.3	3.3	0.0	10	BANSOLTR
1.7	0.0	1.7	10	VERTGRIDQ5
0.0	0.0	0.0	10	BCQ5
0.0	0.0	0.0	10	RHSMULT

The subroutines SPRSCONV, CONVECTION, VERTVEL3_2, and SPRSMLTIN2 collectively use about 22% of the total execution time. This is because they are called for each horizontal node from within the time stepping loop. When called for a particular horizontal node `I`, each of the above subroutines works only on the vertical grid data that are associated with `I`. This indicates that the horizontal node loops involving these subroutines can be executed in parallel.

4. PARALLEL IMPLEMENTATION

The approach used in this project is in the spirit of the Single Program Multiple Data (SPMD) model that is common in programming for distributed memory. The `PARALLEL` and `END PARALLEL` directives were used to enclose the entire initialization and time-stepping portion of the code, including subprogram calls, within a single parallel execution region. The decomposition of work among the threads within the parallel region occurs in the horizontal dimension (i.e., the nodes and elements of the 2-D triangular mesh). During execution in the parallel region, the threads remain in existence and proper data flow is ensured through minimal use of the `BARRIER` synchronization directive. Code that must be executed in serial is handled by the master thread. Since the `BARRIER` can be 30% to 50% less expensive than a `PARALLEL DO`, this approach significantly reduces the amount of overhead associated with OpenMP. This section describes the source code modifications made in QUODDY5 for OpenMP multithreaded processing. Many of the code modifications are similar; therefore, only representative modifications are shown. All modifications can be found by searching the source code on both the `CTJC` and `C$OMP` strings.

The following QUODDY5 program files and subroutines they contain have been modified for multithreaded processing.

quoddy5_1.0_main.f: PROGRAM QUODDY5

quoddy5_1.0_coresubs.f: INITIALIZEQ5, STATIONARYQ5, ELEVATIONQ5, VERTICALQ5, SMAGOR1, QUADSTRESS

quoddy5_1.0_usrsubs_resources.f: UNISIGMAQ5, SINEGRIDQ5

DCMSPAK_000607.f: EQSTATE1_2D, EQSTATE2_2D, RHOXYQ4

NMLPAKS_000607.f: VERTSUM, VERTAVG, SPRSINVMLT, SPRSMLT

To minimize the number of code modifications, the multithreaded subroutines in the quoddy5_1.0_* files retain the same name and calling parameters as the original subroutines. With respect to DCMSPAK and NMLPAKS, separate multithreaded versions have been created. The new routines retain the same name except for a "_MT" suffix added to the end. This choice was made to maintain compatibility of the DCMSPAK and NMLPAKS with other non-OpenMP NML applications, and to allow for calls to these routines from non-OpenMP or master thread regions of QUODDY5. No OpenMP code changes were made to the user-specified subroutines in quoddy5_1.0_usrsubs.f. By restricting the OpenMP code changes in this manner, the user is able to seamlessly switch to the parallel QUODDY5 by compiling the OpenMP code with the appropriate (unmodified) user-specified subroutines.

4.1 Parallel Region

A single parallel region is defined that begins in MAIN just before the initialization section. The code that initiates the parallel region in quoddy5_1.0_main.f is shown here.

```
CTJCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CTJC: Begin OpenMP parallel region
C$OMP PARALLEL DEFAULT(SHARED)
C$OMP+PRIVATE(I,J,DUZDX,DUZDY,DVZDX,DVZDY,DZDX,DZDY)
C$OMP+PRIVATE(VISCX,VISCY,VALMID)
CTJCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

At this point, the team of threads is spawned, with the initial thread being the master or thread number 0. The number of threads in the team is determined by the OMP_NUM_THREADS environment variable set in the shell in which the program executes. By default, all variables declared in MAIN are scoped as shared, i.e., accessible to all threads. Those variables scoped as private to each thread are loop indices, some local scalars, and arrays are used only for data in the vertical direction. The parallel region ends in MAIN just after the time stepping section is finished. The code that finalizes the parallel region in quoddy5_1.0_main.f is shown here.

```
CTJCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CTJC: End OpenMP parallel region
C$OMP END PARALLEL
CTJCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

At this point, all threads except the master thread are terminated. The master thread continues with the program finalization.

The parallel region in QUODDY5 encloses a block of code that includes calls to other sub-routines. According to the OpenMP standard, parameters passed to these subroutines carry the same scope of accessibility by threads that was assigned in the lexical extent of the parallel region (i.e., the code contained directly within the `PARALLEL/END PARALLEL` directive pair). Variables that are defined only within the lexical extent of a subroutine (i.e., local variables) are by default scoped as private. Common blocks and saved variables are automatically scoped as shared. Several QUODDY5 subroutines contained local variables that were required to be scoped as shared. This was handled by placing the local variables in a common block uniquely identified with the enclosing subroutine. For example, the following common block was added to the subroutine `VERTICALQ5`.

```
CTJC: Place some local variables in common to scope them as shared
      COMMON/VERTICALQ5_MT_COM/UNEW,VNEW,SURF
```

Similar uniquely defined common blocks have been added to `INITIALIZEQ5`, `STATIONARYQ5`, `ELEVATIONQ5`, and `RHOXYQ4`. As an alternative to the common block, the Fortran `SAVE` statement could also be used to scope local variables as shared. It is not clear which approach, if any, is preferable.

4.2 Horizontal Node Loops

Most of the computation in QUODDY5 occurs in loops over nodes of the horizontal triangular mesh. In practice, the size of the horizontal dimension will always be larger than the vertical. Therefore, to maintain good scalability, it is necessary to parallelize in the horizontal dimension. To minimize overhead, the choice was made to explicitly partition the horizontal node loops without the use of `OMP DO`. This required that each thread compute and maintain its own horizontal node loop bounds. The horizontal node loop bounds are computed by an equal partition among threads.

In general if the number of threads is N_{th}, then the loop bounds, M_1 and M_2, for thread number I_{th} $(I = 0, ..., N_{th} - 1)$ can be computed as

$$M_1 = I_{th} \left\lfloor \frac{N_2 - N_1 + 1}{N_{th}} \right\rfloor + N_1, \tag{1}$$

$$M_2 = \begin{cases} N_2 & \text{when } I_{th} = N_{th} - 1, \\ (I_{th} + 1) \left\lfloor \frac{N_2 - N_1 + 1}{N_{th}} \right\rfloor + N_1 - 1 & \text{otherwise} \end{cases}, \tag{2}$$

where the original loop bounds are N_1 and N_2.

Equations 1 and 2 are realized in the following subroutine that has been added to the end of quoddy5_1.0_main.f for computing loop bounds for a thread.

```
      SUBROUTINE GET_MT_LOOP_BOUNDS(N1,N2,M1,M2)
      IMPLICIT NONE
      INTEGER ID,NTH,NCH,N1,N2,M1,M2
C$    INTEGER OMP_GET_NUM_THREADS,OMP_GET_THREAD_NUM
C$    EXTERNAL OMP_GET_NUM_THREADS,OMP_GET_THREAD_NUM
      NTH=1
      ID=0
C$    NTH=OMP_GET_NUM_THREADS()
C$    ID=OMP_GET_THREAD_NUM()
      NCH=(N2-N1+1)/NTH
      M1=ID*NCH+N1
      M2=(ID+1)*NCH+N1-1
      IF(ID.EQ.NTH-1) M2=N2
      RETURN
      END
```

Subroutine `GET_MT_LOOP_BOUNDS` takes as input the serial loop bounds (`N1` & `N2`) and computes the new loop bounds (`M1` & `M2`) for the calling thread. Two runtime OpenMP functions are required: `OMP_GET_NUM_THREADS`, which returns the number of threads defined in the encompassing parallel region; and `OMP_GET_THREAD_NUM`, which returns the identifier for the calling thread. The `C$` sentinal indicates to an OpenMP compiler that the executable statement that follows is to be compiled. Non-OpenMP compilers will treat the lines prefixed with `C$` as comments. The local variables `NTH` and `ID` are initialized to the single thread (serial) values. This setup allows the OpenMP QUODDY5 to compile and execute correctly with a non-OpenMP compiler.

At the beginning of the parallel region, each thread makes the following call to determine its own horizontal node loop bounds.

```
      CALL GET_MT_LOOP_BOUNDS(1,NN,INMIN,INMAX)
```

The horizontal nodes are indexed from 1 to `NN`; `INMIN` and `INMAX` are the minimum and maximum horizontal node loop indices for the calling thread. These variables are stored in the following common block that is declared in all subroutines that have been converted to multithreaded processing.

```
      INTEGER INMIN,INMAX
      COMMON/MGMT_MT_COM/INMIN,INMAX
C$OMP THREADPRIVATE(/MGMT_MT_COM/)
```

The `THREADPRIVATE` directive is required wherever this common block is declared. It results in each thread accessing its own private copy of the common block. The alternative to this approach would have been to list `INMIN` and `INMAX` as calling parameters in all subroutines converted to multithreaded processing. However, this alternative would have required changes in some of the user-specified subroutines – something that was avoided in this project.

Within the parallel region and the subroutines converted to multithreaded processing, the horizontal node loops have been modified to use the computed thread loop bounds by replacing oc-curences of `DO I=1,NN` with `DO I=INMIN,INMAX`. For example, the following loops from the vertical structure section of the time-stepping loop,

```
      DO I=1,NN
          ATMxMID(I)=0.5*(ATMxMID(I)+ATMxNEW(I))
          ATMyMID(I)=0.5*(ATMyMID(I)+ATMyNEW(I))
          ATEMPmid(I)=0.5*(ATEMPmid(I)+ATEMPnew(I))
          BTEMPmid(I)=0.5*(BTEMPmid(I)+BTEMPnew(I))
      ENDDO
      DO J=1,NEV
          DO I=1,NN
              SRCmid(I,J)=0.5*(SRCmid(I,J)+SRCnew(I,J))
          ENDDO
      ENDDO
```

have been modified to become

```
CTJC: Horizontal node loop restricted to thread
      DO I=INMIN,INMAX
          ATMxMID(I)=0.5*(ATMxMID(I)+ATMxNEW(I))
          ATMyMID(I)=0.5*(ATMyMID(I)+ATMyNEW(I))
          ATEMPmid(I)=0.5*(ATEMPmid(I)+ATEMPnew(I))
          BTEMPmid(I)=0.5*(BTEMPmid(I)+BTEMPnew(I))
      ENDDO
      DO J=1,NEV
CTJC:     Horizontal node loop restricted to thread
          DO I=INMIN,INMAX
              SRCmid(I,J)=0.5*(SRCmid(I,J)+SRCnew(I,J))
          ENDDO
      ENDDO
```

in the OpenMP QUODDY5. Note that because the thread loop bounds are already computed, no overhead is incurred when executing these loops in parallel. This is true even though the horizontal node loop is nested within the loop over the vertical dimension (`J=1,NEV`). Using `OMP DO` on the nested horizontal node loop would be impractical, since the nested code generated by the OpenMP construct would incur excessive overhead. The alternative would be to reorder the loops so that the vertical is nested within the horizontal. However, this would also incur overhead due to poor utilization of the memory cache because the vertical is the outer dimension of the 2D arrays.

4.3 Synchronization Between Parallel Loops

The association of a thread with a set of horizontal nodes remains fixed in the parallel region. Therefore thread synchronization between successive loops is required only when data conflicts exist between the loops. A data conflict between two loops occurs when the data being accessed by a thread in the second loop are at the same time being modified by another thread still executing in the previous loop. This is illustrated in the following code taken from subroutine `VERTICALQ5`:

```
DO I=1,NN
    SURF(I)=-G*(0.5*(HNEW(I)+HMID(I))-HDOWN(I))
ENDDO
CALL SPRSMLT(PPX,IQ,JQ,SURF,UNEW,NN)
CALL SPRSMLT(PPY,IQ,JQ,SURF,VNEW,NN)
```

where the subroutine `SPRSMLT` is defined as:

```
SUBROUTINE SPRSMLT(QV,IQ,JQ,X,B,NN)
DIMENSION QV(*),X(*),B(*),IQ(*),JQ(*)
KMAX=0
DO I=1,NN
    SUM=0.
    KMIN=KMAX+1
    KMAX=IQ(I)
    DO K=KMIN,KMAX
        SUM=SUM+QV(K)*X(JQ(K))
    ENDDO
    B(I)=SUM
ENDDO
RETURN
END
```

The I'th value of `UNEW` and `VNEW` (`B` in `SPRSMLT`) depends on a range of values of `SURF` (`X` in `SPRSMLT`). This means that a thread computing `UNEW` and `VNEW` for the set of nodes in its domain will require values of `SURF` that may be computed by other threads. To ensure that all of `SURF` is updated prior to computing `UNEW` and `VNEW`, the threads must be synchronized before calling `SPRSMLT`. This is accomplished by inserting a `BARRIER` directive between the `I` loop and the calls to `SPRSMLT`. The multithreaded version of the above example is

```
CTJC: Horizontal node loop restricted to thread
    DO I=INMIN,INMAX
        SURF(I)=-G*0.5*(ZETANEW(I)+ZETAMID(I))
    ENDDO
C$OMP BARRIER
CTJC: Call multithreaded version: SPRSMLT_MT
    CALL SPRSMLT_MT(PPx,IQ,JQ,SURF,Unew,NN)
    CALL SPRSMLT_MT(PPy,IQ,JQ,SURF,Vnew,NN)
```

where the multithreaded subroutine SPRSMLT is defined as

```
    SUBROUTINE SPRSMLT_MT(QV,IQ,JQ,X,B,NN)
CTJC: Variables for thread management
```

```
      INTEGER INMIN,INMAX
      COMMON/MGMT_MT_COM/INMIN,INMAX
C$OMP THREADPRIVATE(/MGMT_MT_COM/)
      DIMENSION QV(*),X(*),B(*),IQ(*),JQ(*)
CTJC: Set proper KMAX for thread domain
      KMAX=0
      IF(INMIN.NE.1) KMAX=IQ(INMIN-1)
CTJC: Loop restricted to thread
      DO 10 I=INMIN,INMAX
      SUM=0.
      KMIN=KMAX+1
      KMAX=IQ(I)
      DO 20 K=KMIN,KMAX
   20 SUM=SUM+QV(K)*X(JQ(K))
   10 B(I)=SUM
      RETURN
      END
```

Note that additional code was added in SPRSMLT_MT to ensure that each thread properly initialized KMAX before entering the 10 loop. Barriers are also used in the code before and after serial regions, as discussed in the next subsection.

4.4 Serial Regions

Calls to user-modifiable subroutines and otherwise non-multithreaded (serial) regions are handled by the master thread (thread id = 0) using the OpenMP MASTER/END MASTER directives. This requires synchronization before the call or serial region to ensure that all threads have updated the data required in the master region. Another synchronization is required after the serial region to ensure that data are updated before the threads continue. The following code from quoddy5_1.0_main.f illustrates how the barriers are used and serial regions are constructed.

```
      DO J=1,NNV
CTJC:     Horizontal node loop restricted to thread
          DO I=INMIN,INMAX
              UZnew(I,J)=0.0
              VZnew(I,J)=0.0
              Tnew(I,J)=0.0
              Snew(I,J)=0.0
              SRCnew(I,J)=0.0
          ENDDO
      ENDDO
CTJC: Let master thread handle point sources
C$OMP BARRIER
C$OMP MASTER
      CALL POINTSOURCEQ5(KDnew,SECnew,ITER,NN,NNV,Zmid,
     &SRCnew,UZnew,VZnew,Tnew,Snew)
C$OMP END MASTER
C$OMP BARRIER
CTJC: Call multithreaded version: VERTSUM_MT
```

```
      CALL VERTSUM_MT(SRCnew,SRCSUMnew,NN,NEV,NNdim)
```

The first `BARRIER` guarantees that each thread finishes zeroing its portion of the *`new` arrays prior to the call to `POINTSOURCEQ5`. The second `BARRIER` guarantees that the master thread is done and all values of `SRCnew` are updated before the threads continue with the call to `VERTSUM`. The `MASTER` `END MASTER` directives define the region where only the master thread functions. All other threads skip the code within the master region and proceed to the first executable statement that follows. In this case, the first executable statement is the `BARRIER`, where the threads will wait until the master thread also reaches the `BARRIER`.

4.5 Other Parallel Constructs

The `OMP DO` directive was also used in QUODDY5 to execute loops in parallel that were not over horizontal nodes (such as loops over horizontal elements) or did not have bounds that matched those used to define the explicit thread loop bounds (such as the boundary conditions in `VERTICALQ5`). The following code from subroutine `STATIONARYQ5` is an example of how this construct was used to execute a horizontal element loop in parallel.

```
CTJC: Horizontal element loop in parallel using OMP DO
C$OMP DO
      DO L=1,NE
         I1=IN(1,L)
         I2=IN(2,L)
         I3=IN(3,L)
         DX(1,L)=X(I2)-X(I3)
         DX(2,L)=X(I3)-X(I1)
         DX(3,L)=X(I1)-X(I2)
         DY(1,L)=Y(I2)-Y(I3)
         DY(2,L)=Y(I3)-Y(I1)
         DY(3,L)=Y(I1)-Y(I2)
         AR(L)=0.5*(X(I1)*DY(1,L)+X(I2)*DY(2,L)+X(I3)*DY(3,L))
         IF(AR(L).LE.0.0)WRITE(2,*)' NEGATIVE AREA IN ELEMENT', L
      ENDDO
C$OMP ENDDO NOWAIT
```

The default static scheduling of threads is used. There is an implicit barrier at the end of a loop associated with an `OMP DO`. When the barrier is not necessary, as in this case, the `NOWAIT` clause is placed at the end of the loop to remove the barrier.

5. VERIFICATION AND PERFORMANCE

Correctness of the parallel program execution has been verified through direct comparison with the original serial program execution for the Yellow Sea regional model (6847 horizontal and 21 vertical nodes) [2]. The verification was done using the full *seasonal* mode in which wind is applied and temperature and salinity are transported prognostically. Since the user-defined output data were of limited precision, verification was done by directly comparing (at full precision) all time-integrated variables using the following process. First, a 10 model day run was executed using the original serial QUODDY5 with a time step of 225 seconds. Every four model hours, the following time-integrated data were written in binary form to a file (tagged with the iteration number):

Hmid(I): Total water column depth;

Umid(I), Vmid(I): Vertically averaged velocity;

Zmid(I,J): Nodal coordinate locations in 3-D;

UZmid(I,J), VZmid(I,J), WZmid(I,J): Nodal values of the X, Y, and Z components of velocity;

Q2mid(I,J), Q2Lmid(I,J): Nodal values of turbulent kinetic energy and the turbulent kinetic energy times the master length scale;

RHOmid(I,J), Tmid(I,J), Smid(I,J): Nodal values of density, temperature, and salinity;

ENZM(I,L), ENZH(I,L), ENZQ(I,L): Elemental values of the vertical diffusivities for momentum, mass variables, and the turbulent variables.

The output was performed near the end of the time stepping loop, after all time integrated variables had been updated. The set of files generated from the original QUODDY5 run provided a baseline for checking the OpenMP QUODDY5 as it was developed. The 10 model day run was repeated with the OpenMP QUODDY5 on different numbers of processors (ranging from 1 to 60). Data written from the OpenMP QUODDY5 runs were compared with the data from the original using a Fortran routine that read the two sets of data and checked for differences. Since the data were written in binary form, the comparison was made at full precision. Possible data conflicts between loops (as described earlier) that were not obvious from studying the program were found by executing the OpenMP QUODDY5 with the number of threads specified greater than the number of processors. This forced threads to contend for resources and disrupted any *natural* thread ordering that might otherwise occur that could hide data conflicts. Since the modifications in the OpenMP QUODDY5 do not alter any of the algorithms or order of numerical operations established in the original program, exact match (bit-for-bit) between the serial and parallel execution has been achieved.

Performance measurements of the OpenMP QUODDY5 were done using both the Yellow Sea model (6847 horizontal nodes) and the Arabian Gulf model (17440 horizontal nodes) with two vertical resolutions of 21 and 51 vertical nodes [7]. During these measurments, transport of temperature and salinity was enabled and file output was disabled. Table 3 lists the results of timing measurments performed on a Sun Enterprise 10000 with 64 Ultra Sparc II 400 MHz processors and 64 GB of memory. The timings are expressed as seconds per model time step. Because the processing on the Sun was not dedicated, each run was repeated 5 times with the minimum time reported in Table 3. The performance of the OpenMP QUODDY5 on a single processor is the same as that of the original serial QUODDY5.

Table 3 — Timing of OpenMP QUODDY5 for Two Horizontal Mesh Sizes, Each with Two Vertical Resolutions

Number of Processors	17440 Horizontal Nodes		6840 Horizontal Nodes	
	21 Vertical	51 Vertical	21 Vertical	51 Vertical
1	9.23	21.10	3.33	8.13
2	4.91	10.86	1.74	4.15
4	2.74	5.78	0.94	2.20
8	1.77	3.59	0.52	1.20
16	1.24	2.25	0.33	0.63
32	0.89	1.39	0.23	0.39

To understand the parallel performance of the OpenMP QUODDY5 it is useful to examine two quantities known as *speedup* and *efficiency*. For a fixed problem size, the speedup on p processors over the single processor execution is defined as $S_p = T_1/T_p$, where T_1 is the serial execution time and T_p is the execution time on p processors. Theoretically, the speedup can never exceed the number of processors. The efficiency, defined as $E_p = S_p/p$, is a measure of the fraction of time for which a processor is usefully employed. In an ideal parallel system and implementation, the speedup is equal to p and the efficiency is equal to one. In practice, the speedup is less than p and efficiency is between zero and one, depending on the design of the parallel system and the parallel program. If we assume that the underlying parallel system is ideal, then the limitations to the scalability of a parallel program can be simply understood by separating its serial and parallel components. Suppose that a parallel program has a remaining serial portion that requires an execution time that is a fraction f of the total single processor execution time. If we assume ideal speedup for the remaining parallel portion of the program, then the overall speedup is given by

$$S_p = \frac{1}{f + \frac{1-f}{p}}. \tag{3}$$

This means that the remaining serial component of the program places an upper bound of $1/f$ on the speedup, no matter how many processors are used.

Figure 2 shows the speedup of the OpenMP QUODDY5 for different mesh sizes as computed from Table 3. We see that for the largest problem size (h=17440, v=51) the OpenMP QUODDY5 on 32 processors is more than 15 times faster than the serial version. This clearly means that with the OpenMP QUODDY5, one has the ability to tackle problem sizes that were not previously practical due to excessively long execution times. One trend that is clear from Fig. 2 is that, as
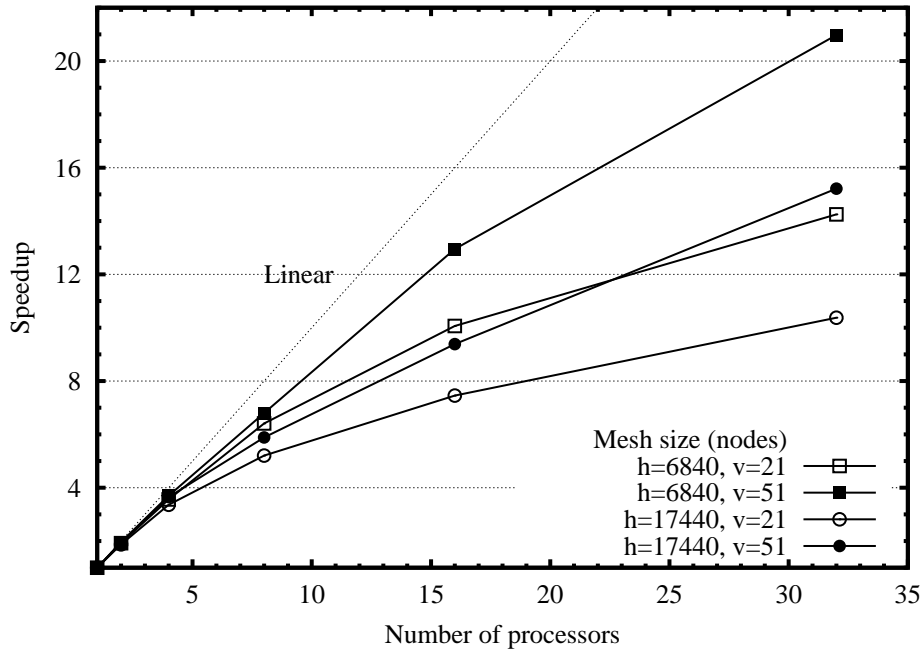

Fig. 2 — Speedup of OpenMP QUODDY5 for Yellow Sea (h=6840) and Arabian Gulf (h=17440) models with two vertical resolutions: v=21 (open points) and v=51 (filled points)

the vertical resolution is increased, the parallel performance improves. This is to be expected, since all of the parallelism is based on the horizontal dimension, and increasing the vertical resolution corresponds to more work at each horizontal node. One unexpected result observed in Fig. 2 is that the overall parallel performance degraded as the horizontal mesh size increased. Performance measurements on other platforms (such as the SGI Origin and the IBM SP) did not exhibit the same behavior. More detailed timing analysis reveals that this is probably due to better cache utilization on the Sun for the smaller mesh size.

Figure 3 shows the corresponding efficiencies as computed from Table 3. For the largest problem size (h=17440, v=51), the efficiency drops to 74% by 8 processors, after which the decrease becomes more gradual. By 32 processors, the efficiency is about 48%. The efficiency is a measure of how well the parallel program utilizes the assigned processors. If one is only interested in elapsed wall time, not resource utilization, then the efficiency is not an issue. In that case, the choice would be made to run on the number of processors that yields the desired turnaround time. However, when resource utilization is an issue, as in the case of allocations based on processor count as well as wall time, one should choose the number of processors such that the efficiency is higher (say above 70%).
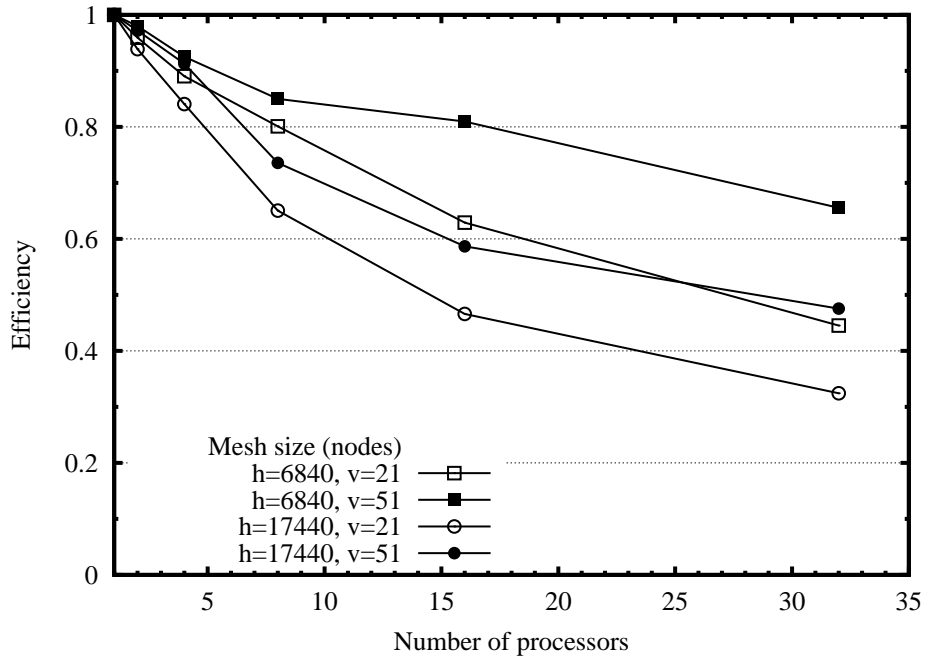

Fig. 3 — Efficiency of OpenMP QUODDY5 for Yellow Sea (h=6840) and Arabian Gulf (h=17440) models with two vertical resolutions: v=21 (open points) and v=51 (filled points)

The logical sections of the time-stepping loop described in Fig. 1 and shown in Appendix B provide a useful approach to further analyze the parallel performance. Figure 4 shows the execution times for each of the logical sections as a function of number of processors. Of the four sections defined, the *wave equation* section is the only one that clearly exhibits serial behavior. (As can be seen from Appendix B2, the wave equation section consists only of the subroutine `ELEVATIONQ5`.) The initial drop in time for the *wave equation* section is because the loops that set up the vertical grid spacing in subroutines called by `VERTGRIDQ5` are executed in parallel. After 4 processors, the

execution time in the *wave equation* section is dominated by `BANSOLTR` and the element loop in `ELEVATIONQ5`, which are completely serial.
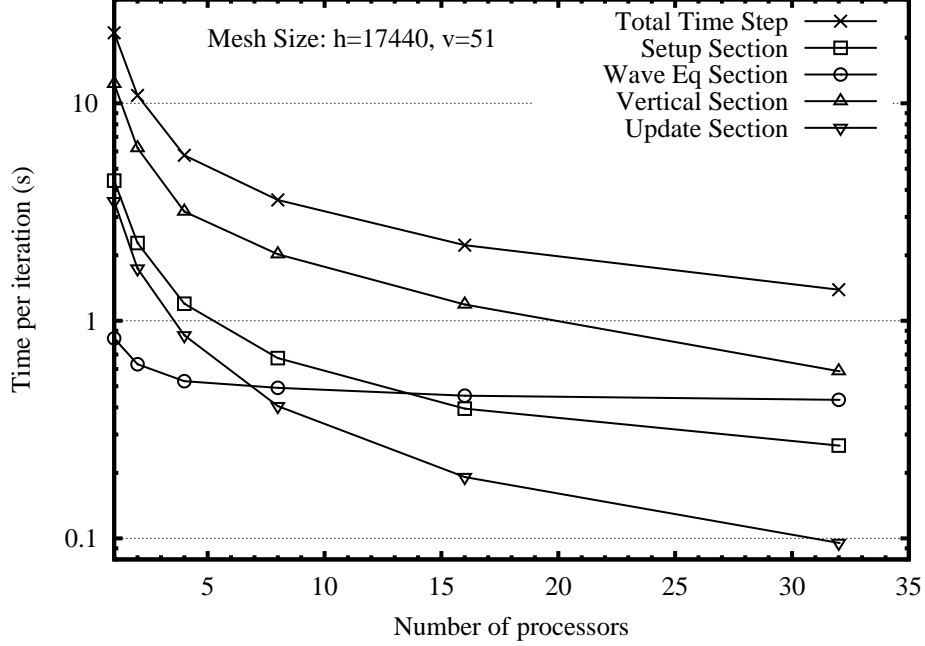

Fig. 4 — Performance of time stepping sections of OpenMP QUODDY5 for the Arabian
Gulf model with a vertical resolution of 51 nodes

Figure 4 also shows that the execution time in the *setup* section begins exhibiting serial behavior after eight processors. A quick look at the code for the *setup* section in Appendix B1 reveals that the scalability will be limited, in part, by `ATMOSQ5` and `POINTSOURCEQ5` because these routines are handled by the master thread. However, for the test cases used in this work there was no atmospheric forcing or point sources. In cases with atmospheric forcing or point sources, the QUODDY5 user could choose to remove the `MASTER/END MASTER` and `BARRIER` directives and implement loops or sections of code within `ATMOSQ5` and `POINTSOURCEQ5` in parallel. The main limitation to the scalability of the *setup* section is the subroutine `SMAGOR1`, which consists of an element loop that must be executed in serial because of data dependencies similar to those in the element loop of `ELEVATIONQ5` (described in Section 3).

Figure 5 shows the speedup for each of the logical sections. One notable feature is that the *update* section exhibits super-linear speedup. This type of behavior occurs in parallel programs because the decrease in the range of loop indices assigned to a thread can result in improved cache utilization. The result of the super-linear speedup of the *update* section at 32 processors is not large: an increase of about 0.2 in the overall speedup. It is disappointing that the speedup of the *vertical structure* section is only about 21 on 32 processors. Detailed timing of the *vertical structure* section shows that the execution time is dominated by a single loop in `VERTICALQ5` in which the vertical structure is computed for vertical grid points directly under each horizontal node. The loop has no data dependencies and should exhibit perfect scaling. In fact, the higher speedups attained for the Yellow Sea mesh are due to near perfect scaling of the *vertical structure* section. This leads us to conclude that the less than desirable performance of the Arabian Gulf model on the Sun E10000 is due to the underlying memory system, not to limitations in the OpenMP QUODDY5 itself.
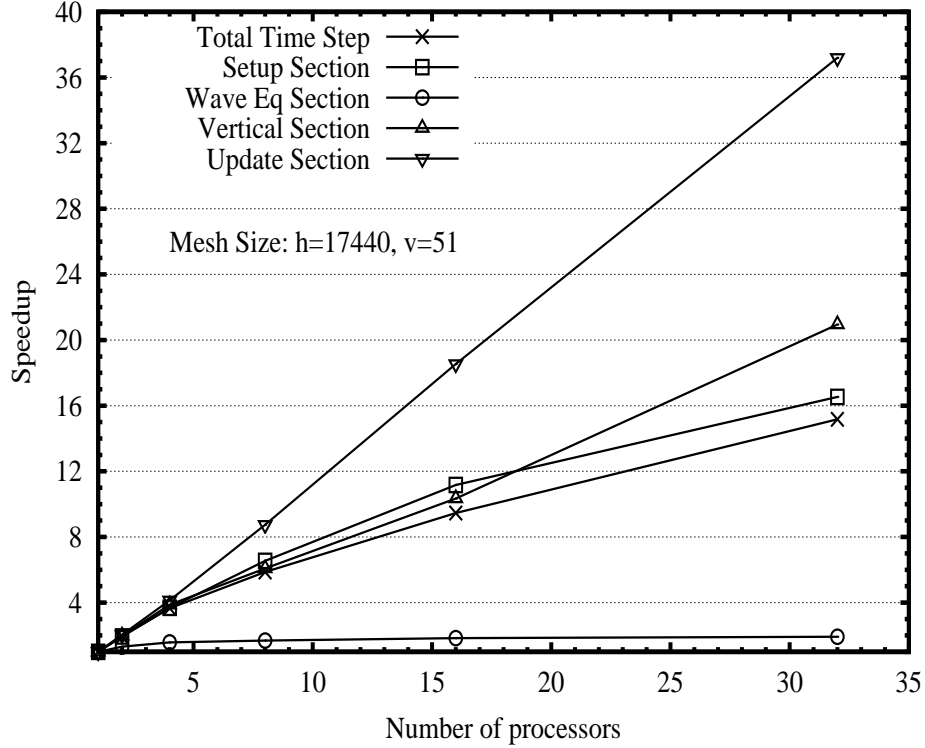
Fig. 5 — Speedup of time stepping sections of OpenMP QUODDY5 for the Arabian Gulf
model with a vertical resolution of 51 nodes

Other than calls to some user-specified routines, the only serial regions that remain in OpenMP QUODDY5 are the element loops in `SMAGOR1` and `ELEVATIONQ5` and the matrix solve routine `BANSOLTR`, which is called from within `ELEVATIONQ5`. We can use the profile data in Tables 1 and 2 to estimate the fraction of execution time that is serial and then use Eq. (3) to compute the expected speedup, assuming that the parallel portion of the program scales perfectly. From Tables 1 and 2, the estimated serial fraction is 7.2% (3.0% for `SMAGOR1`, 0.9% for `ELEVATIONQ5` itself, and 3.3% for `BANSOLTR`). Using Eq. (3), a serial fraction of 7.2% on 32 processors gives an expected speedup of only 9.9, which is much lower than the actual speedup of 15.2 achieved for the Arabian Gulf model (with 51 vertical nodes). As it turns out, a lot of overhead is associated with the profile data obtained from gprof; this causes many of the timings in Tables 1 and 2 to be too large. The gprof profile is still useful as a guide for identifying routines where most of the execution time is spent. Explicit timing of these serial regions in the original QUODDY5 yields the following profile: 0.7% for the `SMAGOR1` element loop, 0.9% for the `ELEVATIONQ5` element loop, and 1.4% for `BANSOLTR`. The newly computed serial fraction of 3.0% for the Arabian Gulf model gives an estimated speedup of 16.6 on 32 processors and an upper bound of 33.3. This analysis indicates that the parallel performance of the OpenMP QUODDY is quite qood and that the only programming limitations to the scalability come from the remaining serial regions.

6. ALTERNATE APPROACHES USING OPENMP

The degree of parallel performance achieved from the SPMD approach described above is because overhead due to OpenMP constructs and reordering of loops has been minimized. The downside to the SPMD approach is that it can require more programming effort than just direct

use of OpenMP directives. In this section, we describe two alternative approaches using OpenMP and compare their parallel performance with the SPMD approach. As it turns out, the extra programming required for the SPMD approach is well worth the effort.

The *minimal* approach to using OpenMP in QUODDY5 is to place directives only on selected computational loops that occupy most of the execution time. This approach does not require any program changes other that inserting a few OpenMP directives. The target regions of code are the nonlinear advection and horizontal diffusion part of the *setup* section, several loops in `RHOXYQ4`, the vertical structure loop in `VERTICALQ5`, and the vertical velocities part of the *update* section. The source code for each of the modifications is listed in Appendix C.

The second alternate approach, which we call *full* OpenMP, is similar to the SPMD approach in that the same single parallel region is used and all the same loops are executed in parallel. The difference is that in the full OpenMP approach the `OMP DO` directive is used, instead of explicit partitioning, for all loops executed in parallel. All loops over horizontal nodes that are nested within a loop over vertical nodes require swapping so that the loop over horizontal nodes is the outer loop. Because the horizontal dimension is the inner dimension on many arrays, this change causes overhead due to poor utilization of cache. Additional overhead is incurred from the `OMP DO` itself.

Figure 6 shows the speedup for each of the programming approaches for the Arabian Gulf model (51 vertical nodes). Although the *minimal* approach requires fewer code modifications, the speedup is severely limited (less than 5 for any number of processors greater than 16). This is a clear demonstration of how the remaining serial portion of a program, in addition to the overhead of thread creation and destruction, can dominate the execution time, even at a moderate number
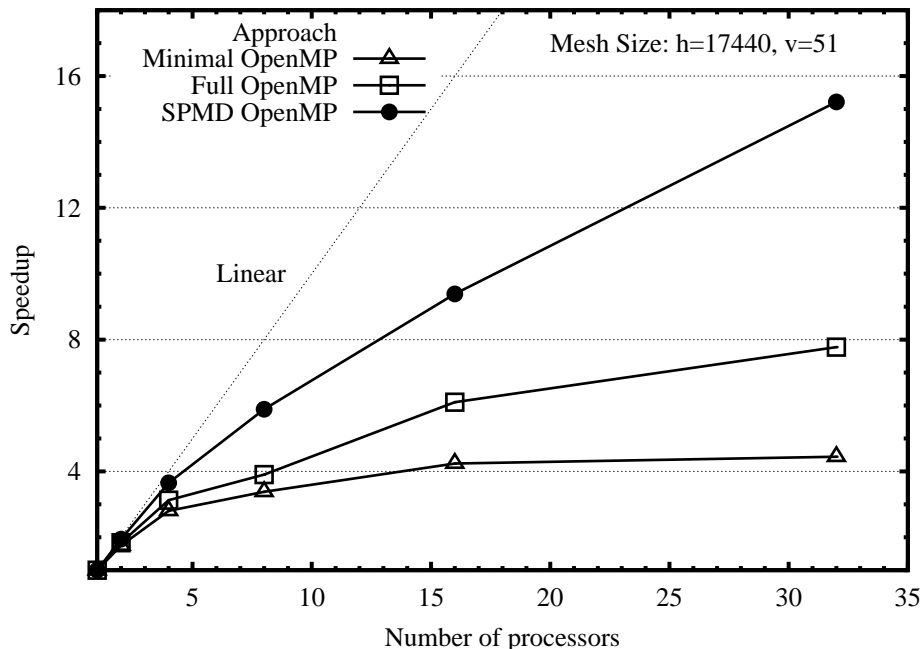

Fig. 6 — Speedup of different OpenMP implementation styles in QUODDY5 for the
Arabian Gulf model with a vertical resolution of 51 nodes

of processors. The *full* approach almost doubles the speedup at 32 processors over the *minimal* approach. However, memory and thread overhead are clearly beginning to dominate, and the speedup of the *full* approach will not rise much above 8. At 32 processors, the speedup of the SPMD approach is still increasing and does not show signs of saturation. These tests clearly show that the SPMD approach provides the ability to capture more of the computation in parallel with less overhead than the other approaches discussed.

7. SUMMARY

This report has presented the development of a parallel version of the 3-D finite-element ocean circulation model known as QUODDY. The model and an execution profile of the original serial code were described. Parallel implementation was accomplished using the OpenMP programming model for shared-memory multiprocessors. The user interface and configuration files remain unchanged, thus providing the possibility for transparent migration of users to the parallel code. Correctness of the parallel program execution was verified by direct comparison at full precision with the original serial program execution in full seasonal mode. Exact match (bit-for-bit) between the serial and parallel execution has been achieved. Performance tests on the Sun E10000 demonstrate that the code is moderately scalable. For the largest problem on hand (17400 nodes in the horizontal mesh, 51 nodes in the vertical mesh), a speedup of 15 over the single-processor execution has been achieved. Better speedups are obtained on other platforms that have lower thread and memory overhead, but these are not presented in this paper. A comparison with alternate approaches using OpenMP was also presented, showing that the approach taking herein provided the best performance. With the new capacity for parallel execution, the time required for high-resolution coastal circulation simulations can be significantly reduced. The viability of applying the model to a new class of problems will aid further developments in coastal ocean circulation modeling.

8. REFERENCES

1. D.R. Lynch, J.T.C. Ip, C.E. Naimie, and F.E. Werner, "Comprehensive Coastal Circulation Model with Application to the Gulf of Maine," *Continental Shelf Res.* **16**(7), 875-906 (1996).

2. C.E. Naimie, C.A. Blain, and D.R. Lynch, "Seasonal Mean Circulation in the Yellow Sea – A Model Generated Climatology," *Continental Shelf Res.* **21**(6-7), 667-695 (2001).

3. D.R. Lynch and W.G. Gray, "A Wave Equation Model for Finite Element Tidal Computations," *Comp. Fluids* **7**, 207-228 (1979).

4. I.P.E. Kinnmark, "The Shallow Water Wave Equations: Formulation, Analysis and Application," Ph.D. dissertation, Department of Civil Engineering, Princeton University, 1985.

5. G.L. Mellor and T. Yamada, "Development of a Turbulence Closure Model for Geophysical Fluid Problems," *Rev. Geophys. Space Phys.* **20**, 851-875 (1982).

6. A.F. Blumberg and G.L. Mellor, "A Description of a Three-Dimensional Coastal Ocean Circulation Model," in *Three-Dimensional Coastal Models*, N.S. Heaps, ed. (American Geophysical Union, Washington, D.C., 1987) Coastal and Estuarine Series **4**, 1-16.

7. C.A. Blain, "Modeling Three-Dimensional, Thermohaline-Driven Circulation in the Arabian Gulf," *Estuarine and Coastal Modeling, Proceedings of the Sixth International Conference*, M.L. Spaulding and H.L. Butler, eds. (American Society of Civil Engineers, 2000), pp. 74-93.

Appendix A

DESCRIPTION OF OPENMP

OpenMP is a parallel programming model for shared memory and distributed shared memory multiprocessor computers. The OpenMP Fortan API consists of compiler directives, which take the form of source code comments and describe the parallelism in the source code. A supporting library of subroutines is also available to applications. The OpenMP specification and related material can be found at the OpenMP web site: http://www.openmp.org. Designed for application developers, Ref. A1 provides a useful introduction to programming with OpenMP.

In Fortran, OpenMP compiler directives (which are treated as comments by a non-OpenMP compiler) have the following possible forms:

```
C$OMP <directive>
!$OMP <directive>
*$OMP <directive>.
```

In fixed-form Fortran source, a directive that contains a character other than a space or a zero in the sixth column is treated as a continuation directive line by the OpenMP compiler. The `PARALLEL` and `END PARALLEL` directive pair constitutes the parallel construct. An OpenMP program begins as a single process, called the *master thread* of execution. When a parallel construct is encountered, a team of threads, with the master thread as the master of the team, is created. The team of threads executes the statements enclosed within the parallel construct, including routines called from within the enclosed statements. At the end of the parallel construct, the threads synchronize and only the master thread remains to continue execution of the program.

The `DO` directive is used within a parallel region to specify that the iterations of the immediately following `DO` loop must be executed in parallel. The iterations of the `DO` loop are divided among the threads according to a `SCHEDULE` clause that may be specified with the `DO` directive. The default schedule specifies that the iterations be divided into equal size *chunks* and statically assigned to threads in the team in a round-robin fashion in the order of the thread number. By default, the number of *chunks* is equal to the number of threads. The following is a simple example that illustrates how the parallel construct and the `DO` directive are used.

```
C$OMP PARALLEL
C$OMP DO
      DO I=2,N
        B(I) = (A(I) + A(I-1)) / 2.0
      ENDDO
C$OMP ENDDO NOWAIT
C$OMP DO
      DO I=1,M
```

```
        C(I) = SQRT(D(I))
      ENDDO
C$OMP ENDDO NOWAIT
C$OMP END PARALLEL
```

There is an implied barrier at the end of a do loop that is parallelized using the `DO` directive. In the above example, the `ENDDO NOWAIT` directive is optional and allows the implied barrier to be avoided.

The `PARALLEL DO` directive provides the application programmer a short cut to specifying a parallel region that contains a single `DO` directive. It is commonly discussed in OpenMP literature and provides a convenient and incremental way to parallelize computationally intensive loops within a program. The above example could be parallelized using the `PARALLEL DO` directive in the following manner.

```
C$OMP PARALLEL DO
      DO I=2,N
        B(I) = (A(I) + A(I-1)) / 2.0
      ENDDO
C$OMP PARALLEL DO
      DO I=1,M
        C(I) = SQRT(D(I))
      ENDDO
```

The downside to this approach is that the creation of threads at the beginning and their subsequent destruction at the end of the loop can require a large number of cycles. The developer must be sure that the loop being parallelized has enough computational work to make the overhead due to the OpenMP constructs worthwhile.

REFERENCE

A1. R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP* (Academic Press, San Diego, 2001).

Appendix B

SOURCE CODE FOR TIME-STEPPING LOOP

This appendix shows the source code for the time-stepping loop from quoddy5_1.0_main.f with OpenMP modifications. The source code is separated into the four logical sections that were used for timing, as described in the Verification and Performance section.

B1. Setup Section

```
C----------------------------------------------------------------------
C SET UP FOR THIS TIME STEP
C
C KDmid,SECmid are the timing parameters for the beginning of the
C current time step => time level K
C Set the timing parameters for the time at the end of the current
C time step        => time level K+1
C
CTJC: Let master thread set timing parameters
C$OMP MASTER
      KDnew=KDmid
      SECnew=SECmid+DelT
      ITER=ITER+1
      IF(SECnew.GE.86400.)CALL UP_DATE(KDnew,SECnew)
C$OMP END MASTER
C
C Zero appropriate arrays and load atmospheric forcing for the end of
C time step (i.e., at time level K+1)
C
CTJC: Horizontal node loop restricted to thread
      DO I=INMIN,INMAX
         ATMxNEW(I)=0.0
         ATMyNEW(I)=0.0
         ATEMPnew(I)=0.0
         BTEMPnew(I)=0.0
         PMEnew(I)=0.0
      ENDDO
CTJC: Let master thread handle atmospheric forcing
C$OMP BARRIER
C$OMP MASTER
      CALL ATMOSQ5(KDnew,SECnew,ITER,NN,NNV,XNOD,YNOD,TMID,SMID,
     &ATMxNEW,ATMyNEW,ATEMPnew,BTEMPnew,PMEnew)
```

```
C$OMP END MASTER
C
C Zero appropriate arrays, load point source information, and compute
C the vertical integral of the volumetric source rate for the end of
C the time step (i.e., at time level K+1). The source information is
C written to the *NEW arrays for efficiency purposes.
C
      DO J=1,NNV
CTJC:    Horizontal node loop restricted to thread
         DO I=INMIN,INMAX
            UZnew(I,J)=0.0
            VZnew(I,J)=0.0
            Tnew(I,J)=0.0
            Snew(I,J)=0.0
            SRCnew(I,J)=0.0
         ENDDO
      ENDDO
CTJC: Let master thread handle point sources
C$OMP BARRIER
C$OMP MASTER
      CALL POINTSOURCEQ5(KDnew,SECnew,ITER,NN,NNV,Zmid,
     &SRCnew,UZnew,VZnew,Tnew,Snew)
C$OMP END MASTER
C$OMP BARRIER
CTJC: Call multithreaded version: VERTSUM_MT
      CALL VERTSUM_MT(SRCnew,SRCSUMnew,NN,NEV,NNdim)
C
C Evaluate the following explicitly for the current time step (i.e., at
C time level K):
C
C   Linearized Bottom Stress Coefficients
C   Baroclinic Pressure Gradients
C   Horizontal Eddy Viscosity/Diffusivity
C   Nonlinear Advection and Horizontal Diffusion of Momentum:
C
C     CONxZ(I,J),CONyZ(I,J) are the 3-D advective plus diffusion terms
C        without FL*DV/DZ => CONxZ=(+UDUDX+VDUDY-Ah*DEL^2 U,ETC).
C     The vertical part is implicit in the tri-diagonal velocity matrix.
C     CONx(I),CONy(I) are the vertical integrals of the advective
C        plus diffusion terms for the wave equation
C        => CONx=INT[CONxZ+(FL+Wmesh)*DU/DZ]dz.
C     Convective terms are turned off along all boundaries by INCONV.
C     NONLIN=0 is implemented by setting INCONV=0 for all nodes.
C
      IF(NLBS.EQ.1)THEN
         CALL QUADSTRESS(aK,Cd,aKMIN,NN,UZmid,VZmid,NNdim)
      ENDIF
```

```
      IF(PRESSURE.EQ.'BAROCLINIC')THEN
CTJC:    Call multithreaded version: RHOXYQ4_MT
         CALL RHOXYQ4_MT(ITER,G,NN,NE,X,Y,IN,PPx,PPy,IQ,JQ,NNV,
     &                   Zmid,RHOmid,NLEV,ZL,BPGx,BPGy,HRBARx,HRBARy,cs)
      ENDIF
C
C Beginning of quoddy4_2.1_P1_main.f modification wrt quoddy4_2.1_main.f
C => new calculation of 2-D horizontal mixing. Note that dimensioning
C    of addition real nodal array SVHmid is required.
C (DRL/JTCI/CEN 10/14/98)
C
      IF(ISMAG.EQ.1)THEN
CTJC:    Horizontal node loop restricted to thread
         DO I=INMIN,INMAX
            HMID(I)=ZETAMID(I)+HDOWN(I)
         ENDDO
         CALL SMAGOR1(AGPGP,Hmid,Umid,Vmid,AR,X,Y,IN,IQ,JQ,NE,SV,NN,
     &                AHI,AH,AHMIN,NFTR,CS)
CTJC:    Horizontal node loop restricted to thread
         DO I=INMIN,INMAX
            SVHmid(I)=SV(I)*Hmid(I)
         ENDDO
CTJC:    Call multithreaded version: SPRSINVMLT_MT
         CALL SPRSINVMLT_MT(AGPGP,IQ,SVHmid,NN)
      ENDIF
C
C End of quoddy4_2.1_P1_main.f modification wrt quoddy4_2.1_main.f.
C
CTJC: Horizontal node loop restricted to thread
      DO I=INMIN,INMAX
         CALL SPRSMLTIN2(I,AGPGP,IQ,JQ,
     &                   UZmid,VISCX,VZmid,VISCY,NNV,NNdim)
         CALL SPRSCONV(I,PPx,PPy,IQ,JQ,UZmid,VZmid,
     &                 Zmid,DUZDX,DUZDY,DVZDX,DVZDY,DZDX,DZDY,NNV,NNdim)
         CALL CONVECTION(I,NNV,UZmid,VZmid,WZmid,Zmid,Zold,DelT,
     &           INCONV,DUZDX,DUZDY,DVZDX,DVZDY,DZDX,DZDY,VISCX,VISCY,
     &           CONxZ,CONyZ,VERTFLX,CONx,CONy,CS)
      ENDDO
```

B2. Wave Equation Section

```
C-----------------------------------------------------------------------
C SOLVE THE WAVE EQUATION
C
C Objectives:Determine total depth at time level    K+1   (=>ZETANEW)
C            and vertical grid at time level         K+1/2 (=>Znew)
C            using known info at time levels         K-1   (=>*OLD)
```

```
C              and                                    K      (=>*MID)
C              with user supplied BC's at various
C              time levels
C
       CALL ELEVATIONQ5(MESHNAME,NN,NE,NHBW,NNV,DelT,G,Tau0,
     &KDnew,SECnew,ITER,DEGLAT,NR,NVN,THETA,XNOD,YNOD,IN,DX,DY,AR,Hdown,
     &ZETAOLD,Uold,Vold,ZETAMID,Umid,Vmid,Zmid,UZmid,VZmid,
     &CONx,CONy,ATMxMID,ATMyMID,aK,HRBARx,HRBARy,
     &SRCSUMold,SRCSUMmid,SRCSUMnew,PMEold,PMEmid,PMEnew,
     &LRVS,NORM,SUMx,SUMy,DSbdry,SUMx1,SUMy1,DSbdry1,QP,SV,SH,DIRICH,
Ccgm-\/
     &ZETANEW,Znew,RAD,CS,COR,TS)
Ccgm-/\
Ccgm IQ, and JQ is needed in conection with the sparse storage of SH
Ccgm+     &ZETANEW,Znew,IQ,JQ,RAD,CS,COR,TS)
Ccgm
```

B3. Vertical Structure Section

```
C-----------------------------------------------------------------------
C SOLVE FOR THE VERTICAL STRUCTURE OF THE 3-D VARIABLES
C
C Objectives: Assemble and solve tridiagonal momentum, Q2, Q2L, T, and S
C             equations at the center of the time step, then convert to
C             values at the end of the time step.
C             => Znew, UZnew, VZnew, Q2new, Q2Lnew, Tnew, and Snew
C                 at time level K+1
C
C Overwrite *MID atmospheric and sourcerate arrays with values at K+1/2
C
CTJC: Horizontal node loop restricted to thread
      DO I=INMIN,INMAX
         ATMxMID(I)=0.5*(ATMxMID(I)+ATMxNEW(I))
         ATMyMID(I)=0.5*(ATMyMID(I)+ATMyNEW(I))
         ATEMPmid(I)=0.5*(ATEMPmid(I)+ATEMPnew(I))
         BTEMPmid(I)=0.5*(BTEMPmid(I)+BTEMPnew(I))
      ENDDO
      DO J=1,NEV
CTJC:     Horizontal node loop restricted to thread
         DO I=INMIN,INMAX
            SRCmid(I,J)=0.5*(SRCmid(I,J)+SRCnew(I,J))
         ENDDO
      ENDDO
C
C Overwrite *NEW dependent variables with source values at K+1/2.
C Increment *SRCmid source term values such that the next time they
C are required, the *SRCmid values will contain the
```

```
C values at time level K for that time step (memory efficiency device).
C Note that *NEW(I,NNV) entries are not altered => the neutrality
C flags are not affected by this evolution.
C
      DO J=1,NEV
CTJC:    Horizontal node loop restricted to thread
         DO I=INMIN,INMAX
             VALMID=UZSRCmid(I,J)
             UZSRCmid(I,J)=UZnew(I,J)
             UZnew(I,J)=0.5*(VALMID+UZnew(I,J))
             VALMID=VZSRCmid(I,J)
             VZSRCmid(I,J)=VZnew(I,J)
             VZnew(I,J)=0.5*(VALMID+VZnew(I,J))
             VALMID=TSRCmid(I,J)
             TSRCmid(I,J)=Tnew(I,J)
             Tnew(I,J)=0.5*(VALMID+Tnew(I,J))
             VALMID=SSRCmid(I,J)
             SSRCmid(I,J)=Snew(I,J)
             Snew(I,J)=0.5*(VALMID+Snew(I,J))
         ENDDO
      ENDDO
C
C Compute vertical structure
C
       CALL VERTICALQ5(NN,NNV,DelT,G,Cd,CLOSURE,MASSVAR,
     &ZLOGBOT,ZLOGTOP,EPSN,KST,NR,NPCN,EPSH,IHHBC,
     &EPSQ,IQADVDIF,IQ2TBC,IQ2BBC,IQ2LTBC,IQ2LBBC,Q2min,Q2Lmin,ELLmin,
     &NORM,LRVS,Hdown,SUMx,SUMy,SUMx1,SUMy1,QP,SV,AGPGP,PPx,PPy,IQ,JQ,
     &INCONV,VERTFLX,aK,BPGx,BPGy,BCFTR,ITER,IUSTARQ2,
     &SRCmid,ATMxMID,ATMyMID,ATEMPmid,BTEMPmid,
     &ZETAMID,Zmid,UZmid,VZmid,Q2mid,Q2Lmid,RHOmid,Tmid,Smid,
     &CONxZ,CONyZ,
     &ZETANEW,Znew,UZnew,VZnew,Q2new,Q2Lnew,Tnew,Snew,
     &EKMMIN,EKHMIN,EKQMIN,ENZM,ENZH,ENZQ,RAD,CS,COR,TS)
```

B4. Update Section

```
C-----------------------------------------------------------------------
C INCREMENT TIMING PARAMETERS AND UPDATE/JUGGLE ARRAYS
C
C Objectives: Increment time such that, for the next time step:
C                 *OLD arrays contain information at time level K-1
C                 *MID arrays contain information at time level K
C
C *OLD<=*MID:
C
      DO J=1,NNV
```

```
CTJC:      Horizontal node loop restricted to thread
           DO I=INMIN,INMAX
               Zold(I,J)=Zmid(I,J)
           ENDDO
       ENDDO
CTJC: Horizontal node loop restricted to thread
      DO I=INMIN,INMAX
          ZETAOLD(I)=ZETAMID(I)
          Uold(I)=Umid(I)
          Vold(I)=Vmid(I)
          PMEold(I)=PMEmid(I)
          SRCSUMold(I)=SRCSUMmid(I)
      ENDDO
C
C *MID<=*NEW:  Thus *MID arrays will contain information for the
C              current time => (KDmid,SECmid), which is time level K
C              for the next time step.
C
CTJC: KDmid & SECmid are handled by master thread
C$OMP MASTER
      KDmid=KDnew
      SECmid=SECnew
C$OMP END MASTER
      DO J=1,NNV
CTJC:      Horizontal node loop restricted to thread
           DO I=INMIN,INMAX
               Zmid(I,J) =Znew(I,J)
               UZmid(I,J)=UZnew(I,J)
               VZmid(I,J)=VZnew(I,J)
               SRCmid(I,J)=SRCnew(I,J)
           ENDDO
       ENDDO
      IF(CLOSURE.EQ.'MY25')THEN
          DO J=1,NNV
CTJC:         Horizontal node loop restricted to thread
              DO I=INMIN,INMAX
                  Q2mid(I,J)=Q2new(I,J)
                  Q2Lmid(I,J)=Q2Lnew(I,J)
              ENDDO
          ENDDO
      ENDIF
      IF(MASSVAR.EQ.'TWO-PROG')THEN
          DO J=1,NNV
CTJC:         Horizontal node loop restricted to thread
              DO I=INMIN,INMAX
                  Tmid(I,J)=Tnew(I,J)
                  Smid(I,J)=Snew(I,J)
```

```
                ENDDO
            ENDDO
CTJC:    Call multithreaded version: EQSTATE2_2D_MT
            CALL EQSTATE2_2D_MT(T0,S0,NN,NNV,TMID,SMID,RHOMID)
        ELSE IF(MASSVAR.EQ.'ONE-PROG')THEN
            DO J=1,NNV
CTJC:        Horizontal node loop restricted to thread
                DO I=INMIN,INMAX
                    Tmid(I,J)=Tnew(I,J)
                ENDDO
            ENDDO
CTJC:    Call multithreaded version: EQSTATE1_2D_MT
            CALL EQSTATE1_2D_MT(T0,NN,NNV,TMID,RHOMID)
        ENDIF
CTJC: Horizontal node loop restricted to thread
        DO I=INMIN,INMAX
            ZETAMID(I)=ZETANEW(I)
            ATMxMID(I)=ATMxNEW(I)
            ATMyMID(I)=ATMyNEW(I)
            ATEMPmid(I)=ATEMPnew(I)
            BTEMPmid(I)=BTEMPnew(I)
            PMEmid(I)=PMEnew(I)
            SRCSUMmid(I)=SRCSUMnew(I)
        ENDDO
C$OMP BARRIER

C Result: Arrays for the *current* time (i.e., the end of the current
C time step) are now stored as *MID
C
C----------------------------------------------------------------------------
C COMPUTE VERTICAL VELOCITIES AND VERTICALLY AVERAGED VELOCITIES
C AT CURRENT TIME (*MID)
C
CTJC: Horizontal node loop restricted to thread
        DO I=INMIN,INMAX
            CALL SPRSCONV(I,PPx,PPy,IQ,JQ,UZmid,VZmid,
     &                     Zmid,DUZDX,DUZDY,DVZDX,DVZDY,DZDX,DZDY,NNV,NNdim)
            CALL VERTVEL3_2(I,NNV,UZmid,VZmid,WZmid,PMEmid,SRCmid,SV,
     &                     Zmid,Zold,DelT,DUZDX,DVZDY,DZDX,DZDY)
        ENDDO
CTJC: Call multithreaded version -- VERTAVG_MT
        CALL VERTAVG_MT(UZmid,Umid,Zmid,NN,NNV,NNdim)
        CALL VERTAVG_MT(VZmid,Vmid,Zmid,NN,NNV,NNdim)
```

Appendix C

SOURCE CODE FOR ALTERNATE MINIMAL OPENMP APPROACH

This appendix lists all source code modifications that were made for the alternate minimal OpenMP approach. The modifications involve only the use of OpenMP directives; no Fortran executable lines were modified. Four regions of the QUODDY5 code were modified: the nonlinear advection and horizontal diffusion part of the *setup* section in the time-stepping loop, several loops in RHOXYQ4, the vertical structure loop in VERTICALQ5, and the vertical velocities part of the *update* section of the time-stepping loop.

The nonlinear advection and horizontal diffusion part of the *setup* section consists of a single loop over horizontal nodes in which the subroutines SPRSMLTIN2, SPRSCONV, and CONVECTION are called for each node. The OpenMP PARALLEL DO directive is used to execute this loop in parallel. The variables scoped as private are the loop index and those that depend only on the vertical grid. The following is the source code for this modification.

```
C$OMP PARALLEL DO DEFAULT(SHARED)
C$OMP+PRIVATE(I,DUZDX,DUZDY,DVZDX,DVZDY,DZDX,DZDY,VISCX,VISCY)
      DO I=1,NN
        CALL SPRSMLTIN2(I,AGPGP,IQ,JQ,
     &                    UZmid,VISCX,VZmid,VISCY,NNV,NNdim)
        CALL SPRSCONV(I,PPx,PPy,IQ,JQ,UZmid,VZmid,
     &                 Zmid,DUZDX,DUZDY,DVZDX,DVZDY,DZDX,DZDY,NNV,NNdim)
        CALL CONVECTION(I,NNV,UZmid,VZmid,WZmid,Zmid,Zold,DelT,
     &            INCONV,DUZDX,DUZDY,DVZDX,DVZDY,DZDX,DZDY,VISCX,VISCY,
     &            CONxZ,CONyZ,VERTFLX,CONx,CONy,CS)
      ENDDO
C$OMP END PARALLEL DO
```

The vertical structure loop in VERTICALQ5 is executed in parallel using the OpenMP PARALLEL DO directive. The default scope for data is made to be private because this is the proper scope for most of the local variables. The variables scoped as shared are the those passed in as arguments to the subroutine and some local variables that are not modified in the vertical structure loop (Unew, Vnew, SURF, USTARQ2, EYE, NEV). The following is the source code for this modification; the details of the vertical structure loop are not included.

```
C$OMP PARALLEL DO DEFAULT(PRIVATE)
C$OMP+SHARED(NN,NNV,DelT,G,Cd,CLOSURE,MASSVAR,ZLOGBOT,ZLOGTOP,EPSN,KST)
C$OMP+SHARED(NR,NPCN,EPSH,IHHBC,EPSQ,IQADVDIF,IQ2TBC,IQ2BBC,IQ2LTBC)
C$OMP+SHARED(IQ2LBBC,Q2min,Q2Lmin,ELLmin,NORM,LRVS,Hdown,SUMx,SUMy)
C$OMP+SHARED(SUMx1,SUMy1,QP,SV,AGPGP,PPx,PPy,IQ,JQ,INCONV,VERTFLX,aK)
C$OMP+SHARED(BPGx,BPGy,BCFTR,ITER,IUSTARQ2,SRC,ATMx,ATMy,ATEMP,BTEMP)
```

```
C$OMP+SHARED(ZETAMID,Zmid,UZmid,VZmid,Q2mid,Q2Lmid,RHOmid,Tmid,Smid)
C$OMP+SHARED(CONxZ,CONyZ,ZETANEW,Znew,UZnew,VZnew,Q2new,Q2Lnew,Tnew)
C$OMP+SHARED(Snew,EKMMIN,EKHMIN,EKQMIN,ENZM,ENZH,ENZQ,RAD,CS,COR,TS)
C$OMP+SHARED(Unew,Vnew,SURF,USTARQ2,EYE,NEV)
      DO 100 I=1,NN
           ...
 ...
 100  CONTINUE
C$OMP END PARALLEL DO
```

In RHOXYQ4, a single parallel region that begins after the initialization is used to enclose the three loops that can be executed in parallel. The default scope for data is private. Variables passed in as arguments and some local variables (RHOL, PPX1, PPY1, DNEIGH) are scoped as shared. The OMP DO directive is used for parallel execution of three loops enclosed in the parallel region. The following is the source code with the modifications to RHOXYQ4.

```
C
C Build nonstandard sprspak arrays on first call
C
      if(ITER.EQ.0)then
         do 5 i=1,nn
            bathy(i)=-z(i,1)
 5       continue
         CALL CALCDNEIGH(nndim,nedim,x,y,in,bathy,nn,ne,DNEIGH)
         CALL BUILDPPX1PPY1(NN,NE,X,Y,IN,BATHY,IQ,JQ,PPX1,PPY1,CS)
      endif
C
C$OMP PARALLEL DEFAULT(PRIVATE)
C$OMP+SHARED(ITER,G,NN,NE,X,Y,IN,PPX,PPY,IQ,JQ,NNV,Z,RHO,NLEV,ZL)
C$OMP+SHARED(RHOX,RHOY,HRBARXE,HRBARYE,CS,RHOL,PPX1,PPY1,DNEIGH)
C
C Interpolate z mesh rho data to level mesh from the top of the
C  level mesh to the level mesh node just above the bottom of the z
C  mesh. Set level mesh values of rho equal to zero for level surfaces
C  below the bottom of the z mesh.
C$OMP DO
      do 10 i=1,NN
           ...
           ...
 10   continue
C
C Begin node loop to compute rhox and rhoy
C$OMP DO
      do 40 i=1,NN
           ...
           ...
 40   continue
C
C Begin element loop to compute hrbarxe and hrbarye
```

```
C$OMP DO
      do 70 k=1,ne
          ...
          ...
 70   continue
C
C$OMP END PARALLEL
C
C End of routine
      return
      end
```

The vertical velocities part of the *update* section consists of a single loop over horizontal nodes followed by two calls to `VERTAVG`. A single `PARALLEL/END PARALLEL` directive pair is used to enclose this region. The horizontal node loop and the loop in `VERTAVG` are executed in parallel using the `OMP DO` directive. The default data scope is set to shared. The variables scoped as private are the loop index and those that depend only on the vertical grid. The modified source code for the *update* section is listed here.

```
C$OMP PARALLEL DEFAULT(SHARED)
C$OMP+PRIVATE(I,DUZDX,DUZDY,DVZDX,DVZDY,DZDX,DZDY)
C$OMP DO
      DO I=1,NN
         CALL SPRSCONV(I,PPx,PPy,IQ,JQ,UZmid,VZmid,
     &                  Zmid,DUZDX,DUZDY,DVZDX,DVZDY,DZDX,DZDY,NNV,NNdim)
         CALL VERTVEL3_2(I,NNV,UZmid,VZmid,WZmid,PMEmid,SRCmid,SV,
     &                  Zmid,Zold,DelT,DUZDX,DVZDY,DZDX,DZDY)
      ENDDO
C$OMP ENDDO NOWAIT
      CALL VERTAVG(UZmid,Umid,Zmid,NN,NNV,NNdim)
      CALL VERTAVG(VZmid,Vmid,Zmid,NN,NNV,NNdim)
C$OMP END PARALLEL
```

Next is the modified source code for the subroutine `VERTAVG`.

```
      SUBROUTINE VERTAVG(F,FAVG,Z,NN,NNV,NNDIM)
      REAL F(NNDIM,*),FAVG(*),Z(NNDIM,*)
C$OMP DO
      DO I=1,NN
         FINT=0.0
         DO J=2,NNV
            FINT=FINT+0.5*(F(I,J)+F(I,J-1))*(Z(I,J)-Z(I,J-1))
         ENDDO
         FAVG(I)=FINT/(Z(I,NNV)-Z(I,1))
      ENDDO
C$OMP ENDDO NOWAIT
      RETURN
      END
```